Lecture 17

Subclasses & Inheritance

Announcements for Today

Reading

- Today: Chapter 18
- Online reading for Thursday
- **Prelim, Nov 6th 7:30-9:30**
  - Material up next Tuesday
  - Review posted next week
  - Recursion + Loops + Classes
- **Conflict with Prelim time?**
  - Submit to Prelim 2 Conflict assignment on CMS
  - Do not submit if no conflict

Assignments

- A4 is due at Midnight
  - Keep reading Piazza
  - Hopefully you just have a few methods left
  - Cannot give extensions
- A5 posted tomorrow
  - Get started immediately!
  - Only one week to do it
  - But short; essentially an extended lab activity

A Interesting Challenge

- How do we add new methods to class `Point`?
  - Open up the `.py` module and add them!
- But Python has many "built-in" classes
  - **Examples:** string, list, time, date (in datatime)
  - **Kivy Examples**: Button, Slider, Image
- What if we want to add methods to these?
  - Where is the module to modify?
  - It is even a good idea to modify it?

Solution: Subclasses

- Class that *extends* another
  - Has attributes, methods from the original class
  - Say it "inherits" these
  - Plus any new ones added
- Original class is parent
  - Also called super class
- Does not have to be in the same module as parent
  - Just import the parent

```
class Employee(object):
    """An Employee with a salary"""
    _name = ''        # a string
    _start = -1       # year; -1 if undef
    _salary = 0.0  # float >= 0
    …

class Executive(Employee):
    """An Employee with a bonus."""
    _bonus = 0.0   # float >= 0
    …
```

Class Definition: Revisited

```
class <name>(<superclass>):
    """Class specification"""
    definitions of fields
    definitions of properties
    constructor (__init__)
    definition of operators
    definition of methods
```

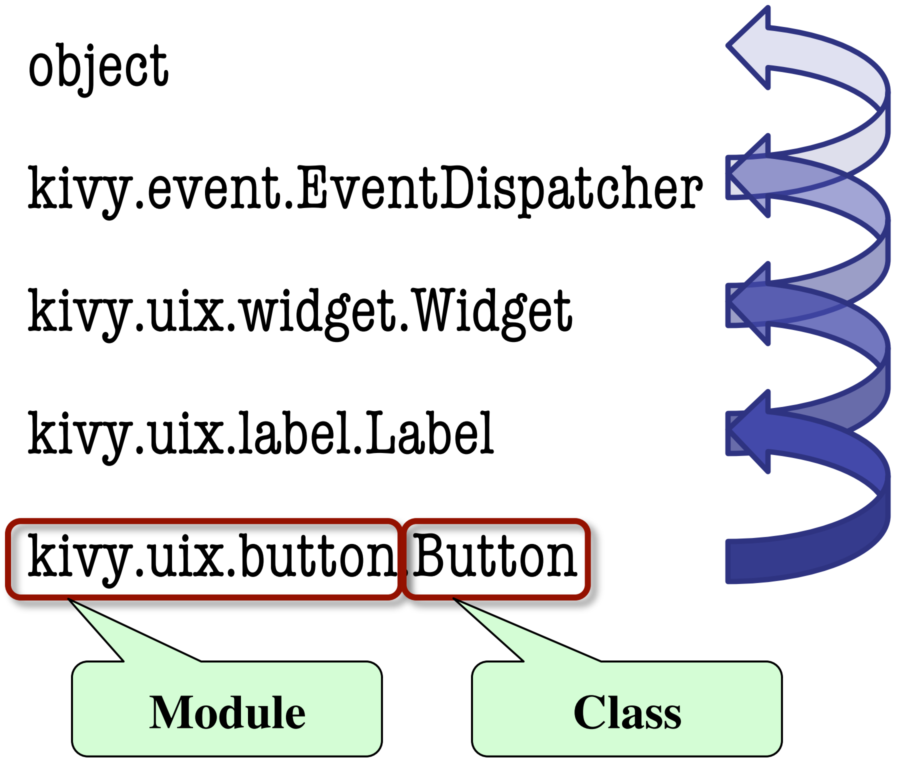
Class type to extend (may need module name)

- Every class must extend *something*
- Previous classes all extended `object`

object and the Subclass Hierarcy

- Subclassing creates a hierarchy of classes
  - Each class has its own super class or parent
  - Until object at the "top"
- object has many features
  - Special built-in fields: __class__, __dict__
  - Default implementations of operators (e.g. __str__)

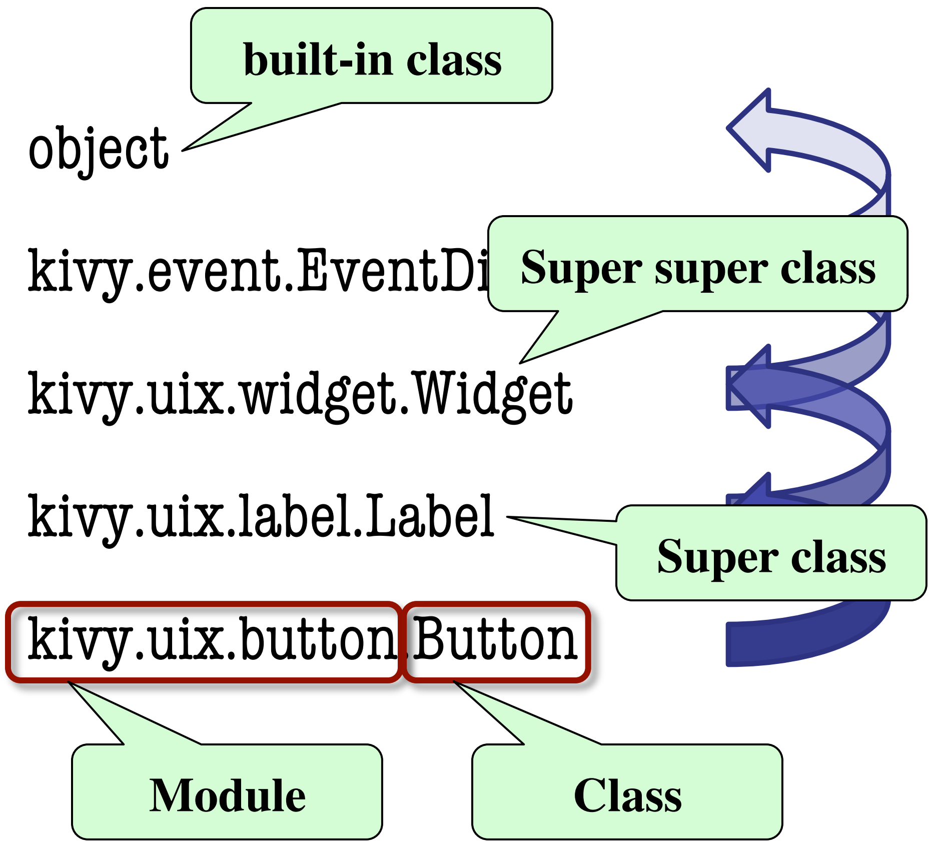
Kivy Example



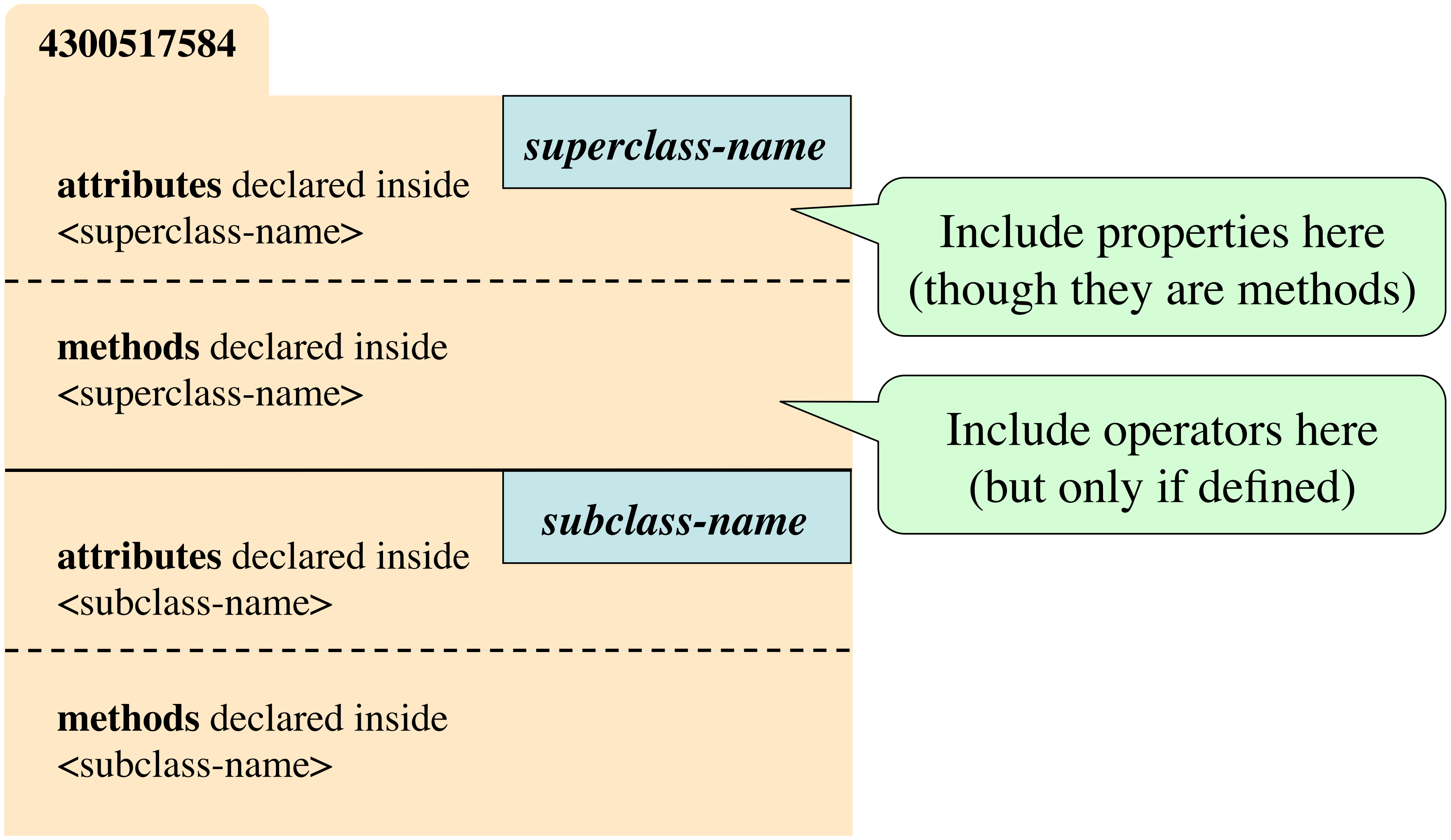
object and the Subclass Hierarcy

- Subclassing creates a hierarchy of classes
  - Each class has its own subclass or parent class
  - Until object at the "top"
- object has many features
  - Special built-in fields: __class__, __dict__
  - Default implementations of operators (e.g. __str__)

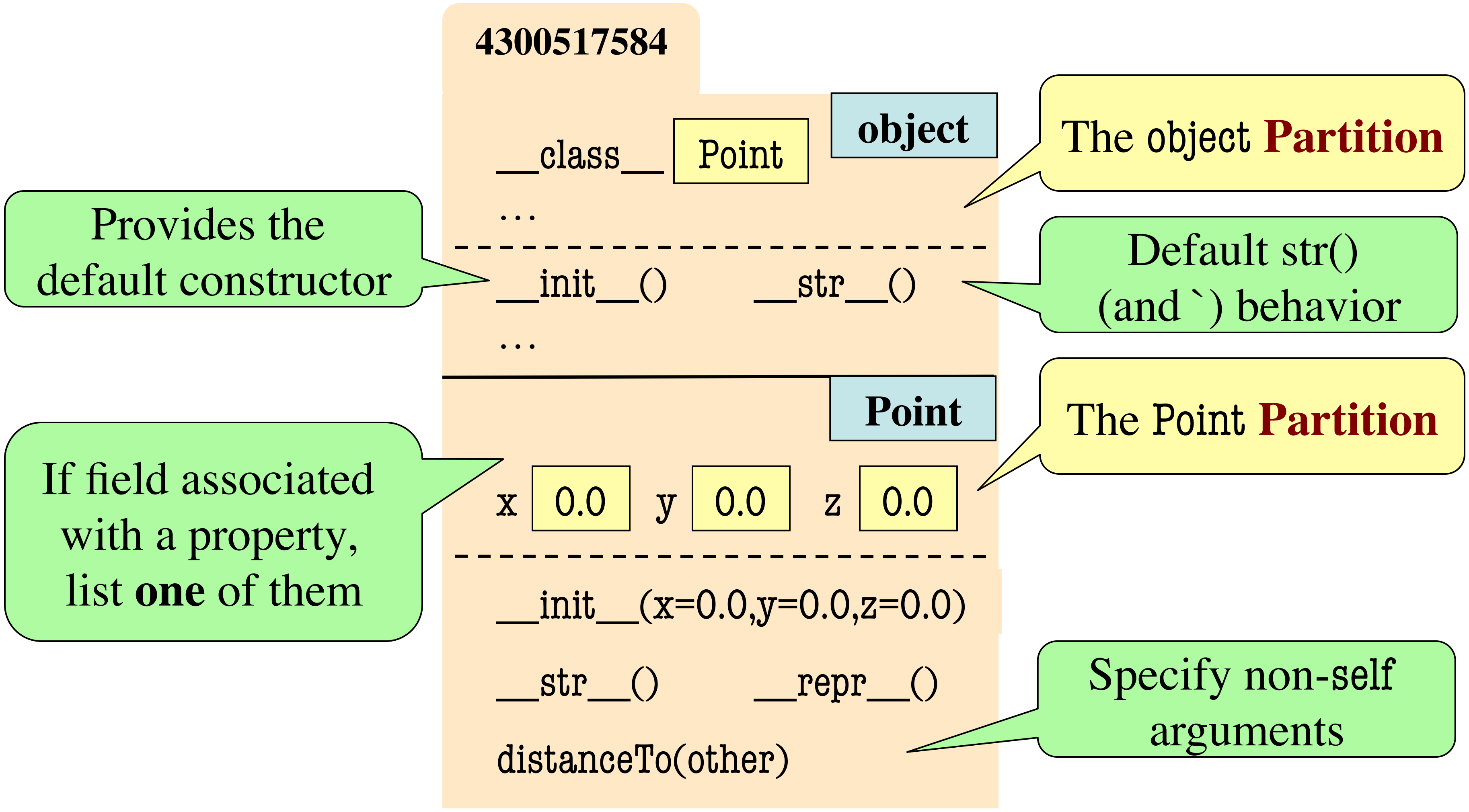
Kivy Example



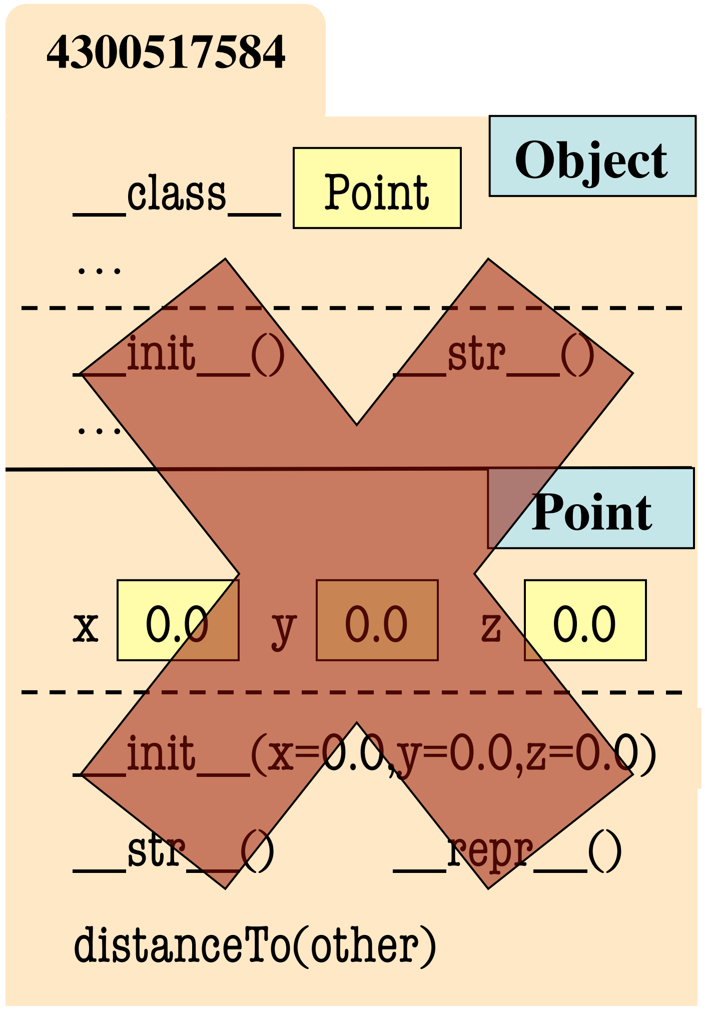
Folder Analogy and Subclasses

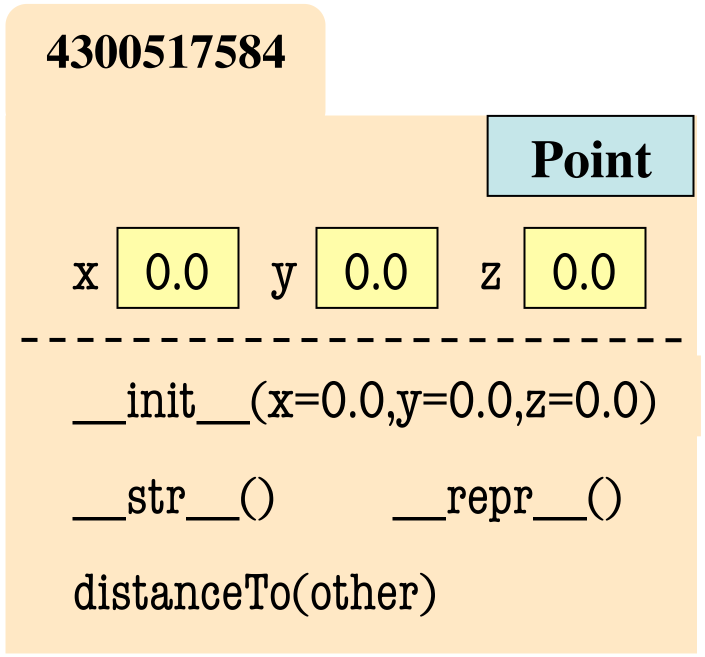

4300517584
superclass-name
attributes declared inside <superclass-name>
methods declared inside <superclass-name>
subclass-name
attributes declared inside <subclass-name>
methods declared inside <subclass-name>
Include properties here (though they are methods)
Include operators here (but only if defined)

Example: Class Point


4300517584
__class__
Point
object
…
__init__()
__str__()
…
Point
x
0.0
y
0.0
z
0.0
__init__(x=0.0,y=0.0,z=0.0)
__str__()
__repr__()
distanceTo(other)
The object Partition
Provides the default constructor
Default str() (and `) behavior
The Point Partition
If field associated with a property, list one of them
Specify non-self arguments

Example: Class **Point**



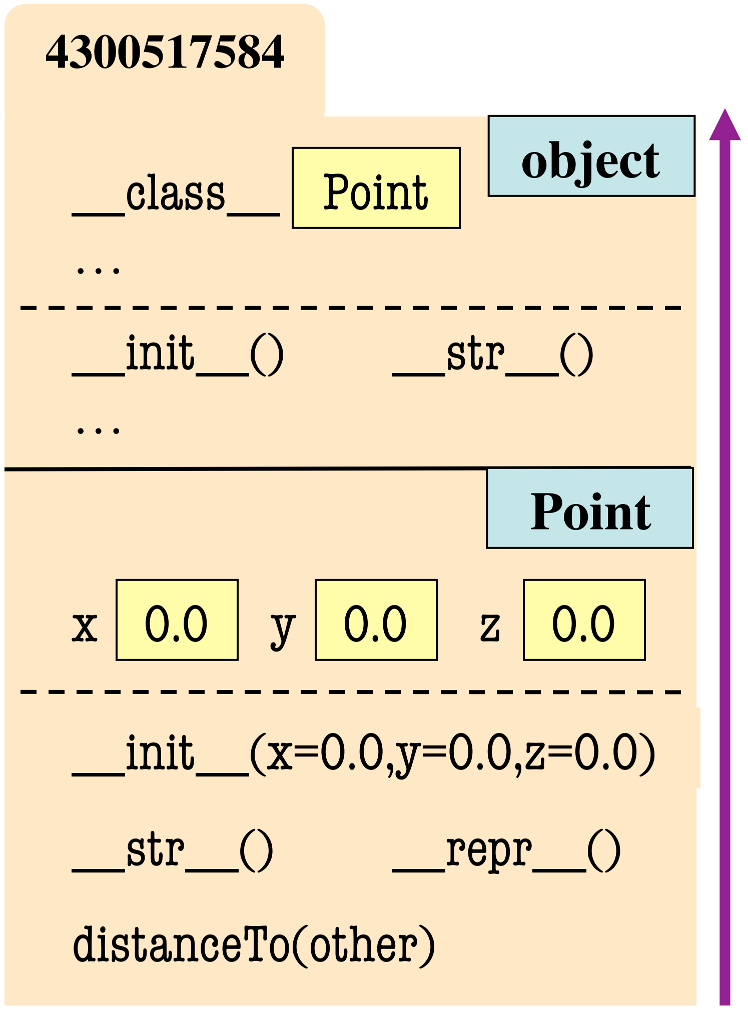


Because it is always there, typically omit the object partition

The Bottom-Up Rule

- Which __str__ does str() use?
  - Work up from bottom of folder
  - Find first method matching name
  - Use that definition
- New method definitions **override** those of parent
- Also applies to
  - Constructor
  - Operators
  - Properties

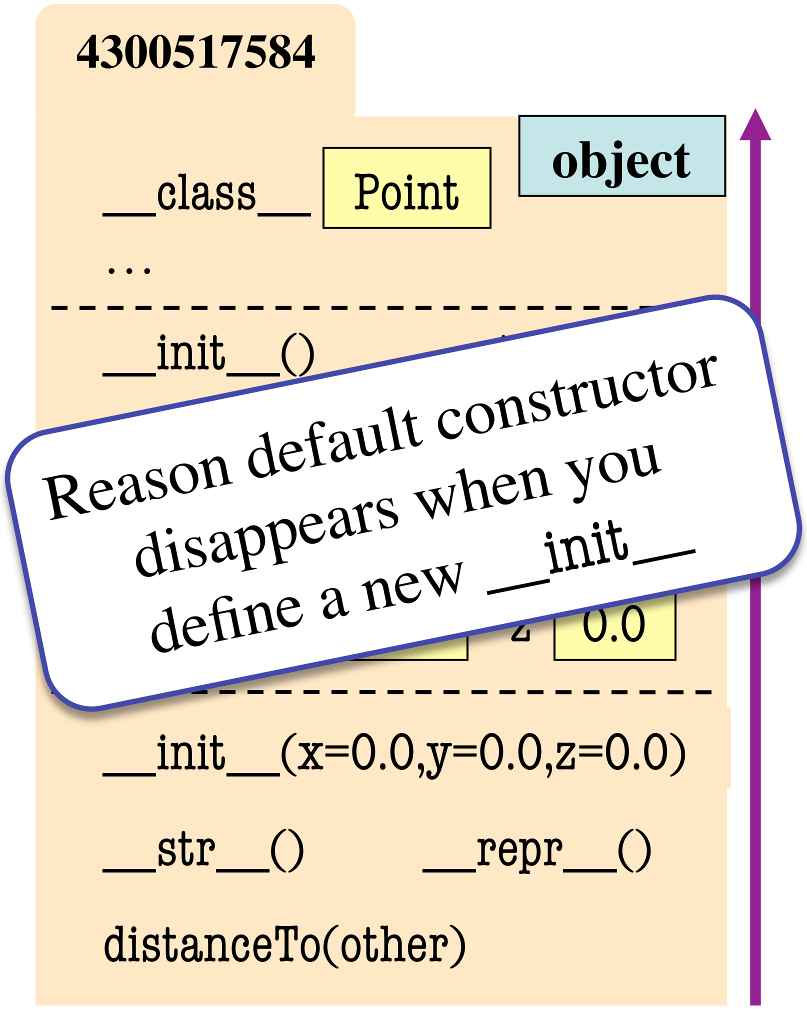
all "methods"



The Bottom-Up Rule

- Which __str__ does str() use?
  - Work up from bottom of folder
  - Find first method matching name
  - Use that definition
- New method definitions **override** those of parent
- Also applies to
  - Constructor
  - Operators
  - Properties

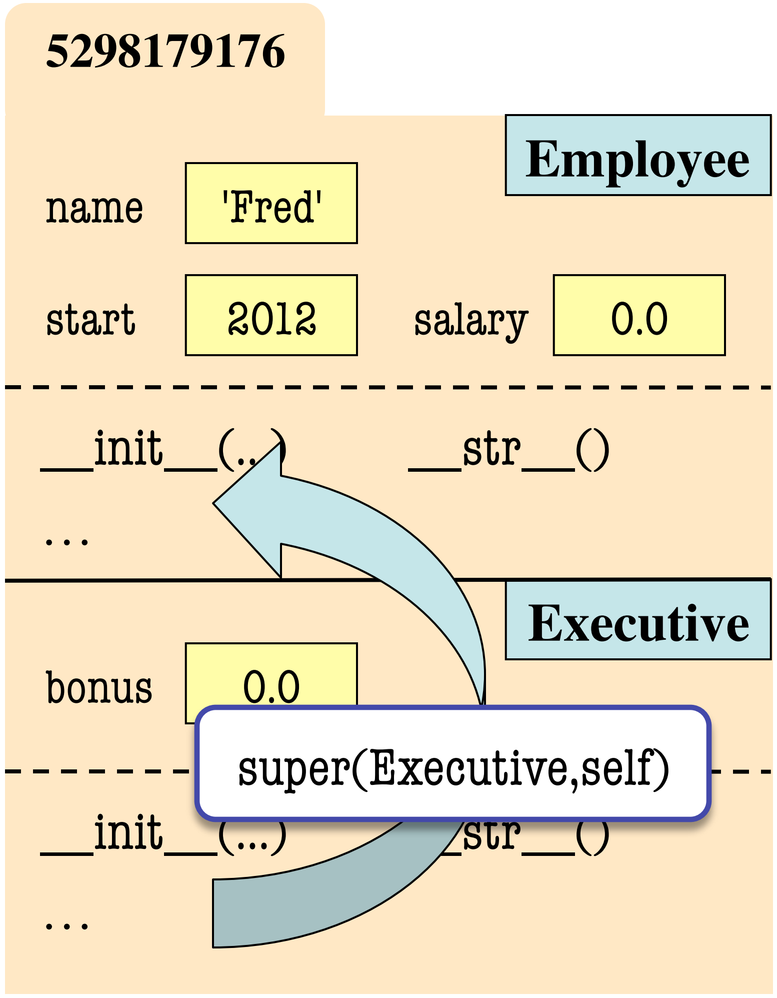
} all "methods"



Reason default constructor disappears when you define a new __init__

Accessing the “Previous” Method

- What if you want definition of the overridden method?
  - New method just *extends*
  - Do not want to repeat code from the old version
- super(<class>,<object>)
  - Returns partition in *object*
  - Parent partition of *class*
- Use it to call a method
  - **Example**: super(Executive,self).__str__()
  - Doesn’t work on properties



Accessing the "Previous" Method

- What if you want definition of the overridden method?
  - New method just *extends*
  - Do not want to repeat code from the old version
- super(<class>,<object>)
  - Returns partition in *object*
  - Parent partition of *class*
- Use it to call a method
  - **Example**: super(Executive,self).__str__()
  - Doesn't work on properties

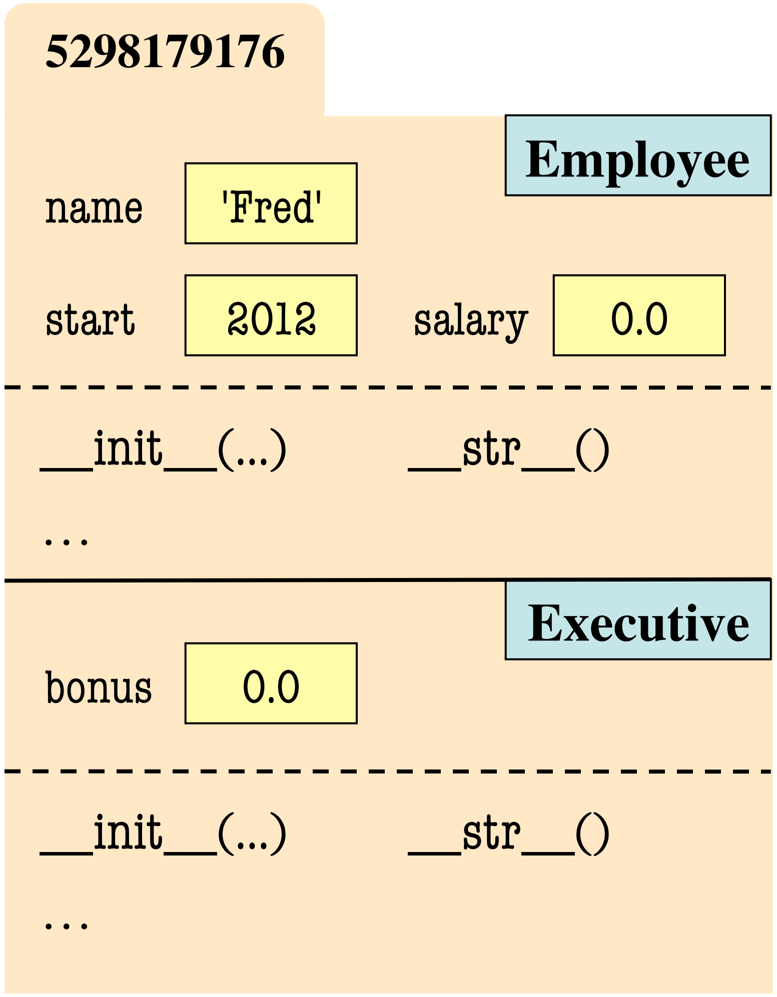
```
class Employee(object):
    """An Employee with a salary"""
    …
    def __str__(self):
        return (self.name +
                ', year ' + str(self.start) +
                ', salary ' + str(self.salary))

class Executive(Employee):
    """An Employee with a bonus."""
    …
    def __str__(self):
        return (super(Executive,self).__str__()
                + ', bonus ' + str(self.bonus) )
```

Primary Application: Constructors

```
class Employee(object):
    …
    def __init__(self,n,d,s=50000.0):
        self._name = n
        self._start = d
        self._salary = s
```

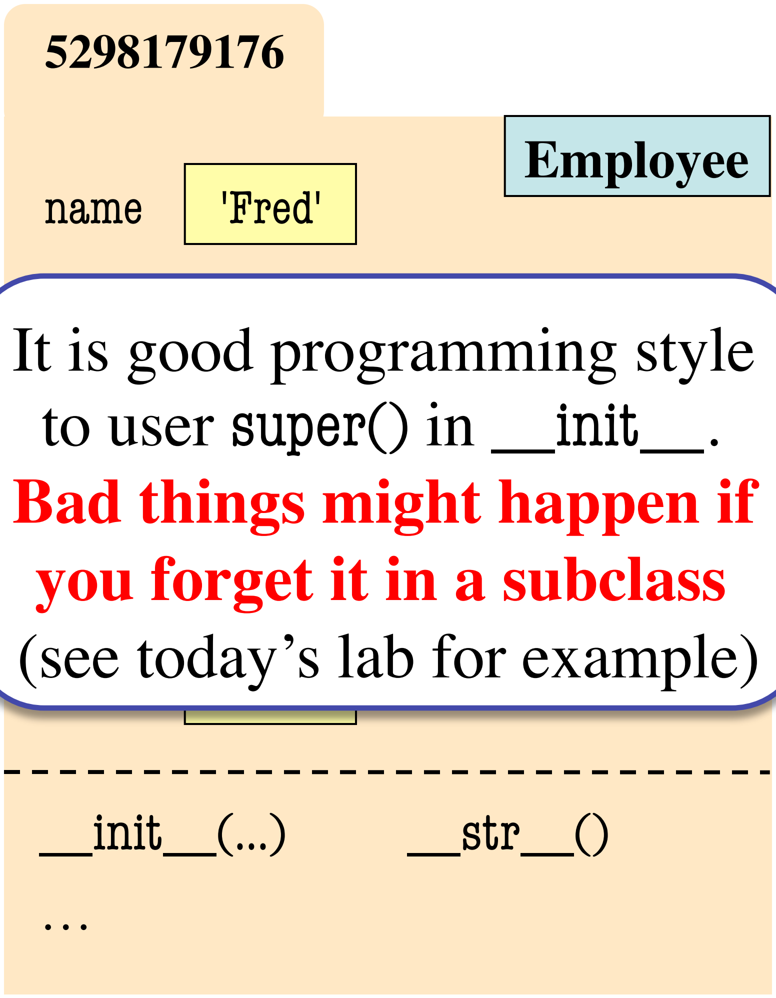
```
class Executive(Employee):
    …
    def __init__(self,n,d,b=0.0):
        super(Executive,self).__init__(n,d)
        self._bonus = b
```



Primary Application: Constructors

```
class Employee(object):
    ...
    def __init__(self,n,d,s=50000.0):
        self._name = n
        self._start = d
        self._salary = s
```

```
class Executive(Employee):
    ...
    def __init__(self,n,d,b=0.0):
        super(Executive,self).__init__(n,d)
        self._bonus = b
```



It is good programming style to user super() in __init__.
Bad things might happen if you forget it in a subclass
(see today's lab for example)

Properties and Inheritance

- Properties: all or nothing
  - Typically inherited
  - Or fully overridden (both getter and setter)
- When override property, **completely** replace it
  - Cannot use super()
- Very rarely overridden
  - **Exception**: making a property read-only
  - See employee.py

```
class Employee(object):
    …
    @property
    def salary(self):
        return self._salary

    @salary.setter
    def salary(self,value):
        self._salary = value


class Executive(Employee):
    …
    @property  # no setter; now read-only
    def salary(self):
        return self._salary
```